

Change Impact Analysis for Aspect-Oriented Software Evolution

Jianjun Zhao

Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0295, Japan
zhao@cs.fit.ac.jp

ABSTRACT

Change impact analysis is a useful technique for software evolution. Many techniques have been proposed for supporting change impact analysis of procedural or object-oriented software, but no effort has been made for change impact analysis of aspect-oriented software. In this paper, we present an approach to supporting change impact analysis of aspect-oriented software based on *program slicing* technique. The main feature of our approach is to assess the effect of changes in an aspect-oriented program by analyzing its source code, and therefore, the process of change impact analysis can be automated completely.

1. INTRODUCTION

Software change is an essential operation for software evolution. The change is a process that either introduces new requirements into an existing system, or modifies the system if the requirement were not correctly implemented, or moves the system into a new operation environment. The mini-cycle of change as described in [11] is composed of several phases: *request for change*, *planning phase* consisting of program comprehension and change impact analysis, *change implementation* including restructuring for change and change propagation, *verification and validation*, and *re-documentation*. Among these phases, in this paper we focus our attentions on the issue of planning phase, especially, change impact analysis to support aspect-oriented software evolution.

Change impact analysis is the task that through which the programmers can assess the extent of the change, i.e., the software component that will impact the change, or be impacted by the change. Change impact analysis provides techniques to address the problem by identifying the likely ripple-effect of software changes and using this information to re-engineer the software system design.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWPE 2002 Orlando Florida
Copyright ACM 2002 1-58113-545 -9/02/05...\$5.00

From the viewpoint of *separation of concern* in software development [4], we may consider to perform change impact analysis at many levels of software systems during software evolution, for example, at the specification, design, architecture, or code level. We may also perform change impact analysis on different languages and design paradigms, for example, on procedural languages or object-oriented languages. Moreover, change impact analysis must adapt to the emergence of new software development methods, and change impact analysis techniques for new languages and design paradigms must be defined based on models that are relevant to these new paradigms.

Recently, aspect-oriented programming has been proposed as a technique for improving separation of concerns in software design and implementation [5]. Aspect-oriented programming works by providing explicit mechanisms for capturing the structure of crosscutting concerns in software systems. Aspect-oriented programming languages can be used to cleanly modularize the crosscutting structure of concerns such as exception handling, synchronization, performance optimizations, and resource sharing, that are usually difficult to express cleanly in source code using existing programming techniques. Aspect-oriented programming languages can control such code tangling and make the underlying concerns more apparent, making programs easier to develop and maintain. As research in aspect-oriented programming is reaching maturity with a number of active research products, it is necessary to perform change impact analysis on aspect-oriented software because it allows one to capture the change effect information of the software so that one can perform software evolution actions on aspect-oriented software.

In an aspect-oriented system, the basic program unit is an aspect rather than a procedure or a class. An aspect with its encapsulation of state with associated advice (operations) is a significantly different abstraction than the procedure units within procedural programs or class units within object-oriented programs. The inclusion of join points in an aspect further complicates the static relations among aspects and classes. Therefore, in order to perform change impact analysis on aspect-oriented software, models that are appropriate for representing aspect-oriented systems are needed.

However, most work on change impact analysis focused on procedural or object-oriented software [3, 6, 7, 8, 13] as well

```

ce0 public class Point {
e1     protected int x, y;
me2     public Point(int _x, int _y) {
e3         x = _x;
e4         y = _y;
    }
me5     public int getX() {
e6         return x;
    }
me7     public int getY() {
e8         return y;
    }
me9     public void setX(int _x) {
e10        x = _x;
    }
me11    public void setY(int _y) {
e12        y = _y;
    }
me13    public void printPosition() {
e14        System.out.println("Point at("+x+", "+y+")");
    }
me15    public static void main(String[] args) {
e16        Point p = new Point(1,1);
e17        p.setX(2);
e18        p.setY(2);
    }

ce19 class Shadow {
e20     public static final int offset = 10;
e21     public int x, y;

me22     Shadow(int x, int y) {
e23         this.x = x;
e24         this.y = y;
me25     public void printPosition() {
e26         System.out.println("Shadow at
            ("+x+", "+y+")");
    }
}

ase27 aspect PointShadowProtocol {
e28     private int shadowCount = 0;
me29     public static int getShadowCount() {
e30         return PointShadowProtocol.
            aspectOf().shadowCount;
    }

e31     private Shadow Point.shadow;
me32     public static void associate(Point p, Shadow s){
e33         p.shadow = s;
    }
me34     public static Shadow getShadow(Point p) {
e35         return p.shadow;
    }

pe36     pointcut setting(int x, int y, Point p):
        args(x,y) && call(Point.new(int,int));
pe37     pointcut settingX(Point p):
        target(p) && call(void Point.setX(int));
pe38     pointcut settingY(Point p):
        target(p) && call(void Point.setY(int));

ae39     after(int x, int y, Point p) returning :
        setting(x, y, p) {
e40         Shadow s = new Shadow(x,y);
e41         associate(p,s);
e42         shadowCount++;
    }

ae43     after(Point p): settingX(p) {
e44         Shadow s = new getShadow(p);
e45         s.x = p.getX() + Shadow.offset;
e46         p.printPosition();
e47         s.printPosition();
    }

ae48     after(Point p): settingY(p) {
e49         Shadow s = getShadow(p);
e50         s.y = p.getY() + Shadow.offset;
e51         p.printPosition();
e52         s.printPosition();
    }
}

```

Figure 1: A sample AspectJ program.

as software architectures [9, 12], and there exists no study of change impact analysis for aspect-oriented software until now. In this paper, we present an approach to supporting change impact analysis of aspect-oriented software based on *program slicing* technique. The main feature of our approach is to assess the effect of changes in an aspect-oriented program by analyzing its source code, and therefore, the process of change impact analysis can be automated completely.

Program slicing, originally introduced by Weiser [10], is a decomposition technique which extracts program elements related to a particular computation. A *program slice* consists of those parts of a program that may directly or indirectly affect the values computed at some program point of interest, referred to as a *slicing criterion*. The task to compute program slices is called *program slicing*. As shown in [2], program slicing is an essential technique to support change impact analysis of procedural or object-oriented software. Using program slicing to support change impact analysis of aspect-oriented software promises benefit for aspect-oriented software evolution. When a maintenance programmer wants to modify a statement in an aspect-oriented program in or-

der to satisfy new requirements, the programmer must first investigate which statements will affect or be affected by the modified statement. By using a slicing tool, the programmer can extract the parts of the paper containing those statements that might affect, or be affected by, the modified statement. The slicing tool which provides such change impact information can assist the programmer greatly.

The rest of the paper is organized as follows. Section 2 briefly introduces the AspectJ. Section 3 shows a motivation example. Section 4 introduces some notions about slicing aspect-oriented software. Section 5 shows how to perform change impact analysis for aspect-oriented software. Concluding remarks are given in Section 6.

2. ASPECT-ORIENTED PROGRAMMING WITH ASPECTJ

We assume that readers are familiar with the basic concepts of aspect-oriented programming, and in this paper, we use AspectJ [1] as our target language to show the basic idea of our change impact analysis for aspect-oriented software.

Below, we use a sample program taken from [1] to briefly introduce the AspectJ. The program shown in Figure 1 associates shadow points with every `Point` object and contains one `PointShadowProtocol` aspect that stores a shadow object in every `Point` and two classes `Point` and `Shadow`.

AspectJ is a seamless aspect-oriented extension to Java. AspectJ adds some new concepts and associated constructs to Java. These concepts and associated constructs are called join points, pointcut, advice, introduction, and aspect.

The *join point* is an essential element in the design of any aspect-oriented programming language since join points are the common frame of reference that defines the structure of crosscutting concerns. The join points in AspectJ are well-defined points in the execution of a program. The join points in AspectJ are *method or constructor call*, *method or constructor execution*, *class or object initialization*, *field reference or assignment*, and *handler execution* [1].

A *pointcut* is a set of join points that optionally exposes some of the values in the execution of those join points. AspectJ defines several primitive *pointcut designators* that can identify all types of join points. Pointcuts in AspectJ can be composed and new pointcut designators can be defined according to these combinations. For example, In aspect `PointShadowProtocol` three pointcuts are declared with the names of `setting`, `settingX`, and `settingY`.

Advice is used to define some code that is executed when a pointcut is reached. AspectJ provides three types of advice, that is, *before*, *after*, and *around*. For example, In aspect `PointShadowProtocol`, there are three *after* advice `setting`, `settingX`, and `settingY`. Advice declarations can change the behavior of classes they crosscut, but can not change their static type structure. For crosscutting concerns that can operate over the static structure of type hierarchies, AspectJ provides forms of introduction.

Introduction in AspectJ can be used by an aspect to add new fields, constructors, or methods (even with bodies) into given interfaces or classes. Introduction can be public or private, where a private introduction means only code in the aspect that declared it can refer or access the introduced fields, constructors, or methods. For example, In aspect `PointShadowProtocol`, introduction declaration `private Shadow Point.shadow;` privately introduces a field named `shadow` of type `Shadow` in `Point`. This means that only code in the aspect can refer to `Point`'s `shadow` field.

Aspects are modular units of crosscutting implementation. Aspects are defined by aspect declarations, which have a similar form of class declarations. Aspect declarations may include advice, pointcut, and introduction declarations as well as other declarations such as method declarations, that are permitted in class declarations. For example, the program in Figure 1 defines one aspect named `PointShadowProtocol`.

An AspectJ program can be divided into two parts: *non-aspect code* which includes some classes, interfaces, and other language constructs as in Java, and *aspect code* which includes aspects for modeling crosscutting concerns in the pro-

gram. Any implementation of AspectJ is to ensure that the aspect and non-aspect code run together in a properly coordinated fashion. Such a process is called *aspect weaving* and involves making sure that applicable advice runs at the appropriate join points. For detailed information about AspectJ, one can refer to [1].

3. A MOTIVATION EXAMPLE

We present a simple example to explain our approach on change impact analysis of aspect-oriented software using program slicing.

Consider the AspectJ program *P* shown in Figure 1. Suppose a maintenance programmer needs to modify the statement `ce2 (return y;)` of *P* in order to add some new functions to *P*. The first thing the programmer has to do is to investigate the effect of the change, i.e., which statements may be affected, or affect the statement `ce2` through variable *y*. A common way is to manually check the source code of the program to find such information. However, it may be very time-consuming and error-prone since there may be complex dependence relations between `ce2` and other statements in *P*. However, if the programmer has a slicer at hand, the work may probably be simplified and automated without the disadvantages mentioned above. In such a scenario, the slicer is invoked, which takes as input: (1) a statement `ce2`, and (2) a variable *y* in `ce2`, (this is a slicing criterion). The slicer then computes a backward and forward slice of *P* respectively with respect to the criterion and outputs the slices to the programmer. A backward slice is a set of statements of *P* that might affect the value of *y* at `ce2`, and a forward slice is also a set of statements of *P* that might be affected by the value of *y* at `ce2`. The other parts of *P* that might not affect or be affected by *y* will be removed, i.e., sliced away from *P*. The programmer can thus examine only those statements included in the slices to investigate the impact of modification.

4. PROGRAM SLICING FOR ASPECT-ORIENTED SOFTWARE

In this section, we introduce some notions about static slicing of an aspect-oriented program. The more detailed definitions can be found in [14].

In analyzing changes for an aspect-oriented program, we usually want to know the answers of some questions such as "which statements might affect a statement in an aspect-oriented program?" and "which statements might be affected by a statement in an aspect-oriented program?" In order to answer these questions, we can define some notions about static slicing of an aspect-oriented program.

A *static slicing criterion* for an aspect-oriented program is a tuple (s, v) , where *s* is a statement in the program and *v* is a variable used at *s*, or a call called at *s*. A *static backward slice* $SBS(s, v)$ of an aspect-oriented program on a given slicing criterion (s, v) consists of all statements in the program that might possibly affect the value of the variable *v* at *s* or the value returned by the call *v* at *s*. A *static forward slice* $SFS(s, v)$ of an aspect-oriented program on a given slicing criterion (s, v) consists of all statements in the program that might possibly be affected by the value of the

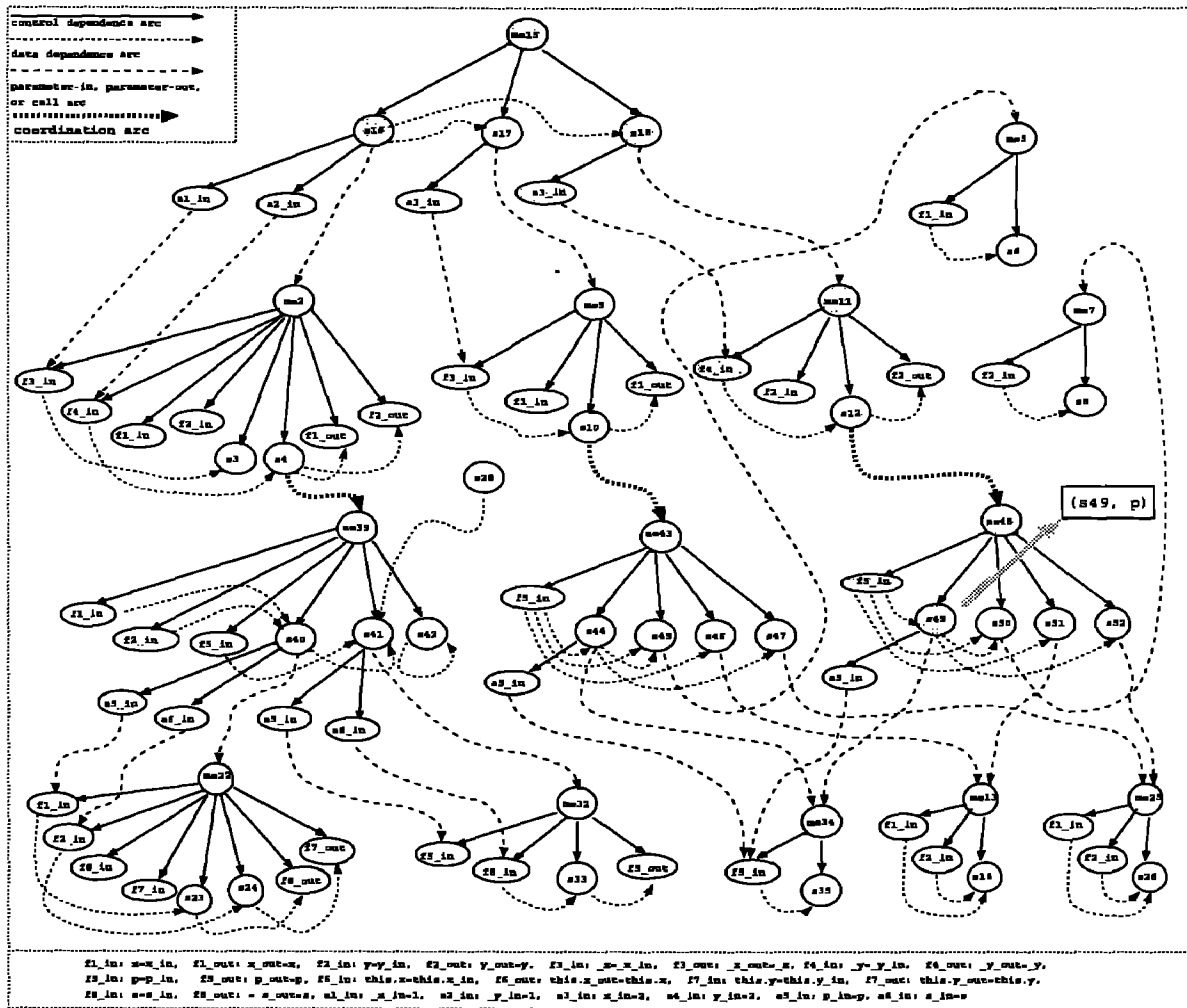


Figure 2: An ASDG of the program in Figure 1 and a slice of the program on slicing criterion (s49, p).

variable v at s or the value returned by the call v at s .

5. CHANGE IMPACT ANALYSIS OF ASPECT-ORIENTED SOFTWARE

Roughly speaking, the process of change impact analysis of an aspect-oriented program is how to find some backward and forward slices of the program by starting from a statement being changed. In [14] we presented a two-phase algorithm to find a static slice of an aspect-oriented program based on its aspect-oriented system dependence graph. In this section we first introduce the aspect-oriented system dependence graph briefly, and then describe the slicing algorithm. For more detailed information, one can refer to [14].

5.1 Aspect-Oriented System Dependence Graphs

The *aspect-oriented system dependence graph* (ASDG) [14] of an aspect-oriented program is a collection of module dependence graphs each representing a module¹ in an aspect or a class of the program, and some additional arcs to represent direct or indirect dependencies between a call and the called module and transitive interprocedural data dependencies.

The *module dependence graph* (MDG) of a module is a di-graph whose vertices represent statements or predicate expressions in the module and arcs represent two types of dependence relationships, i.e., *control dependence*, and *data dependence*. Control dependence represents control conditions on which the execution of a statement or expression depends in the module. Data dependence represents the data flows between statements in the module. Each MDG

¹in this paper we use *module* to stand for a single advice, introduction, or method in an aspect or a class of an aspect-oriented program

has a unique vertex called *module start vertex* to represent the entry of the module.

To model parameter passing, formal parameter vertices are created to associate with each module start vertex. There is a *formal-in vertex* for each formal parameter of the module and a *formal-out vertex* for each formal parameter that may be modified by the module. Also, a *call vertex* and actual parameter vertices are created to associate with each call site. There is an *actual-in vertex* for each actual parameter and an *actual-out vertex* for each actual parameter that may be modified by the called module. In addition, each formal parameter vertex is control dependent on the module start vertex, and each actual parameter vertex is control dependent on the call vertex.

The construction of the complete ASDG can be performed by connecting MDGs at call sites. A *call dependence arc* which represents the call relationships is added between the call vertex of the calling module's MDG and the start vertex of the called module's MDG. Actual-in and formal-in vertices are connected by *parameter-in dependence arcs* and formal-out and actual-out vertices are connected by *parameter-out dependence arcs*. These parameter arcs can represent parameter passing. Moreover, to represent the *transitive flow of dependencies* in the ASDG, *summary dependence arcs* [3] are created by connecting an actual-in vertex to an actual-out vertex if the value associated with the actual-in vertex affects the value associated with the actual-out vertex.

Example. Figure 2 shows the complete ASDG of the sample AspectJ program in Figure 1.

5.2 Computing Slices for Aspect-Oriented Programs

Since the ASDG proposed for an aspect-oriented program can be regarded as an extension of the SDG proposed by Larsen and Harrold for object-oriented software [7], we can use the two-pass slicing algorithm proposed in [3] to compute a static slice of an aspect-oriented program based on its ASDG.

In the first step, the algorithm traverses backward along all arcs except parameter-out arcs, and set marks to those vertices reached in the ASDG, and then in the second step, the algorithm traverses backward from all vertices having marks during the first step along all arcs except call and parameter-in arcs, and sets marks to reached vertices in the ASDG. The slice is the union of the vertices of the ASDG marked during the first and second steps. Similar to the backward slicing described above, we can also apply the forward slicing algorithm [3] to the ASDG to compute a forward slice of an aspect-oriented program.

Example. Figure 2 shows a backward slice which is represented in shaded vertices and computed with respect to the slicing criterion (s49, p).

6. CONCLUDING REMARKS

In this paper, we presented an approach to supporting change impact analysis of aspect-oriented software based on *program slicing* technique. The main feature of our approach is

to assess the effect of changes in an aspect-oriented program by analyzing its source code, and therefore, the process of change impact analysis can be automated completely.

To demonstrate the usefulness of our impact analysis approach, we plan to implement a slicing tool for AspectJ and perform some experiments using the slicing tool to show the effectiveness of our slicing-based change impact analysis approach in supporting aspect-oriented software evolution.

7. REFERENCES

- [1] The AspectJ Team, "The AspectJ Programming Guide," 2001.
- [2] S. A. Bohner and R. S. Arnold, "Software Change Impact Analysis," IEEE Computer Society Press, 1996.
- [3] S. Horwitz, T. Reps and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transaction on Programming Language and System*, Vol.12, No.1, pp.26-60, 1990.
- [4] W. L. Hursch and C. V. Lopes, "Separation of Concern," Technical Report, 1995.
- [5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *proc. 11th European Conference on Object-Oriented Programming*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.
- [6] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Yoyoshima, and C. Chen, "Change Impact Identification in Object-Oriented Software Maintenance," *Proc. International Conference on Software Maintenance*, pp.202-211, 1994.
- [7] L. D. Larsen and M. J. Harrold, "Slicing Object-Oriented Software," *Proceeding of the 18th International Conference on Software Engineering*, German, March, 1996.
- [8] J. P. Loyall and S. A. Mathisen, "Using Dependence Analysis to Support the Software Maintenance Process," *Proc. International Conference on Software Maintenance*, 1993.
- [9] J. A. Stafford and A. L. Wolf, Architecture-Level Dependence Analysis for Software Systems, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 11, No. 4, pp.431-453, 2001.
- [10] M. Weiser, "Program Slicing," *IEEE Transaction on Software Engineering*, Vol.10, No.4, pp.352-357, 1984.
- [11] S. S. Yau, J. S. Collofello, and T. MacGregor, "Ripple Effect Analysis of Software Maintenance," *Proc. of the COMPSAC'78*, pp.60-65, IEEE Computer Society Press, 1978.
- [12] J. Zhao, "Applying Slicing Technique to Software Architectures," *Proc. 4th IEEE International Conference on Engineering of Complex Computer Systems*, pp.87-98, August 1998.
- [13] J. Zhao, "Slicing Concurrent Java Programs," *Proc. Seventh IEEE International Workshop on Program Comprehension*, pp.126-133, May 1999.
- [14] J. Zhao, "Slicing Aspect-Oriented Software," *Proc. 10th IEEE International Workshop on Program Comprehension*, Paris, French, June 27-29, 2002. (to appear)